Universidade Estadual de Londrina

PEDRO ANTÔNIO MESSIAS LEMOS

DESENVOLVIMENTO E AVALIAÇÃO DE UM FUZZER PARA TESTES DE SEGURANÇA EM APIS REST

LONDRINA

2023

PEDRO ANTÔNIO MESSIAS LEMOS

DESENVOLVIMENTO E AVALIAÇÃO DE UM FUZZER PARA TESTES DE SEGURANÇA EM APIS REST

Trabalho de Conclusão de Curso apresentado ao Curso de Bacharelado em Ciência da Computação do Departamento de Computação da Universidade Estadual de Londrina, como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação.

Orientador: Prof(a). Dr(a). Bruno Bogaz Zarpelão

LONDRINA

2023

Ficha de identificação da obra elaborada pelo autor, através do Programa de Geração Automática do Sistema de Bibliotecas da UEL

Sobrenome, Nome.
Título do Trabalho : Subtitulo do Trabalho / Nome Sobrenome. - Londrina, 2017.
100 f. : il.

Orientador: Nome do Orientador Sobrenome do Orientador.
Coorientador: Nome Coorientador Sobrenome Coorientador.
Dissertação (Mestrado em Ciência da Computação) - Universidade Estadual de Londrina, Centro de Ciências Exatas, Programa de Pós-Graduação em Ciência da Computação, 2017.
Inclui bibliografia.

1. Assunto 1 - Tese. 2. Assunto 2 - Tese. 3. Assunto 3 - Tese. 4. Assunto 4 - Tese. I. Sobrenome do Orientador, Nome do Orientador. II. Sobrenome Coorientador, Nome Coorientador. III. Universidade Estadual de Londrina. Centro de Ciências Exatas. Programa de Pós-Graduação em Ciência da Computação. IV. Título.

PEDRO ANTÔNIO MESSIAS LEMOS

DESENVOLVIMENTO E AVALIAÇÃO DE UM FUZZER PARA TESTES DE SEGURANÇA EM APIS REST

Trabalho de Conclusão de Curso apresentado ao Curso de Bacharelado em Ciência da Computação do Departamento de Computação da Universidade Estadual de Londrina, como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação.

BANCA EXAMINADORA

Orientador: Prof(a). Dr(a). Bruno Bogaz Zarpelão
Universidade Estadual de Londrina

Prof. Dr. Segundo Membro da Banca
Universidade/Instituição do Segundo Membro da Banca – Sigla instituição

---

Prof. Dr. Terceiro Membro da Banca
Universidade/Instituição do Terceiro Membro da Banca – Sigla instituição

---

Prof. Ms. Quarto Membro da Banca
Universidade/Instituição do Quarto Membro da Banca – Sigla instituição

Londrina, 24 de abril de 2023.

AGRADECIMENTOS

LEMOS, P. A. M.. **Desenvolvimento e Avaliação de um Fuzzer para Testes de Segurança em APIs REST**. 2023. 26f. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) – Universidade Estadual de Londrina, Londrina, 2023.

RESUMO

Com o aumento na utilização de APIs REST em sistemas contemporâneos, garantir a segurança dessas interfaces tornou-se um desafio crítico. Um *fuzzer* é uma ferramenta projetada para testar a segurança, introduzindo entradas aleatórias em um programa para identificar vulnerabilidades. No entanto, muitas ferramentas de *fuzzing* enfrentam limitações quando aplicadas às APIs, incluindo inadequação ao protocolo HTTP, dificuldades com mecanismos de autenticação e autorização, e desafios na manipulação de estruturas de dados complexas. Adicionalmente, a identificação precisa de erros é desafiadora devido à ambiguidade nas respostas das APIs, uso de códigos de erro personalizados, e a possibilidade de erros silenciosos ou mascarados, tornando a detecção de vulnerabilidades uma tarefa complexa. Diante dessa problemática, este trabalho foca no desenvolvimento de um fuzzer especializado para APIs REST. O objetivo é identificar erros e vulnerabilidades através da análise de respostas da API, considerando especificidades como endpoints, argumentos e códigos de retorno esperados. O fuzzer também abordará desafios como respostas ambíguas, autenticação e validações específicas.

Palavras-chave:

LEMOS, P. A. M.. **Development and Evaluation of a Fuzzer for Testing REST APIs**. 2023. 26p. Final Project (Bachelor of Science in Computer Science) – State University of Londrina, Londrina, 2023.

ABSTRACT

With the rising use of REST APIs in contemporary systems, ensuring the security of these interfaces has become a critical challenge. A fuzzer is a tool designed to test security by introducing random inputs into a program to identify vulnerabilities. However, many fuzzing tools face limitations when applied to APIs, including inadequacy with the HTTP protocol, challenges with authentication and authorization mechanisms, and difficulties in handling complex data structures. Additionally, the precise identification of errors is challenging due to the ambiguity in API responses, the use of custom error codes, and the possibility of silent or masked errors, making vulnerability detection a complex task. In light of these issues, this project focuses on the development of a specialized fuzzer for REST APIs. The aim is to identify errors and vulnerabilities through the analysis of API responses, taking into account specifics such as endpoints, arguments, and expected return codes. The fuzzer will also address challenges like ambiguous responses, authentication, and specific validations.

Keywords:

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

No mundo contemporâneo, as tecnologias da web permeiam diversos aspectos das nossas vidas cotidianas. À medida que a dependência de aplicações web intensifica, as Interfaces de Programação de Aplicativos (APIs) assumem um papel crucial, servindo como o elo que permite a comunicação eficaz entre diferentes programas [1]. Entretanto, o incremento na complexidade das APIs também resulta no aumento de riscos de segurança associados a elas [2].

Deste modo, a segurança de APIs emerge como um tópico de grande relevância e interesse na comunidade científica e no setor industrial [3]. Há uma diversidade de métodos empregados para salvaguardar a segurança das APIs, os quais incluem a análise estática de código, a implementação de firewalls de aplicativos da web e a utilização de *scanners* de vulnerabilidades. O *fuzzing* — uma técnica de teste automatizado que envolve o fornecimento de entradas malformadas, aleatórias ou semi-aleatórias a um programa — tem se mostrado eficaz na identificação de vulnerabilidades até então desconhecidas [4].

Contudo, ferramentas de *fuzzing* originalmente não foram desenvolvidas com um foco específico em APIs Web [5]. Já que possuem características intrínsecas, como mecanismos de autenticação, limitação de taxa e *logs* complexos, que muitas vezes não são adequadamente abordados por *fuzzers* tradicionais [6]. Portanto, é importante reconhecer que, devido a esses desafios, tais ferramentas frequentemente não atingem um nível de adequação satisfatório para testes nesse tipo de APIs [6].

O presente trabalho é motivado pela necessidade de desenvolver um *fuzzer* focado na avaliação de segurança de APIs REST. O objetivo central é criar e avaliar uma ferramenta que possa simular ataques de maneira sofisticada, considerando as peculiaridades das APIs e as limitações que *fuzzers* tradicionais possuem, tais como lidar com autenticação, analisar códigos de resposta, fornecer uma forma eficiente e personalizável de identificar vulnerabilidades e erros, bem como gerenciar diferentes formatos de entrada (parâmetros, JSON, etc.). Os experimentos e avaliações propostos serão conduzidos no CIAAgro UEL (Centro de Inteligência Artificial no Agro da UEL).

Este trabalho de conclusão de curso está organizado da seguinte maneira:

- Capítulo 1 — Introdução: descrição do tema abordado de maneira breve e concisa, bem como uma explicação da estrutura utilizada;
- Capítulo 2 — Fundamentação Teórica: Define e explica conceitos essenciais para a compreensão do trabalho, e também os conceitos utilizados nos procedimentos metodológicos para se alcançar os resultados desejados. Os conceitos definidos e

detalhados são:

  - APIs;
  - APIs REST;
  - Segurança de APIs REST;
  - Testes de segurança;
  - Fuzzing;
  - Fuzzing de APIs;
  - Ferramentas de fuzzing;

- Capítulo 3 —

2 FUNDAMENTAÇÃO TEÓRICA

2.1 APIs

As Interfaces de Programação de Aplicativos (APIs, do inglês *Application Programming Interfaces*) são conjuntos de protocolos, rotinas e ferramentas que permitem a comunicação e interação entre diferentes sistemas de software. Elas abrangem uma vasta gama de soluções, incluindo bibliotecas, frameworks, serviços web e outros códigos reutilizáveis, evidenciando a diversidade e versatilidade dessas interfaces no mundo do desenvolvimento [7]. Conforme a definição fornecida pela AWS [8], uma API atua como uma porta através da qual diferentes serviços de software podem interagir, permitindo que um sistema solicite informações de outro e receba uma resposta em um formato padronizado.

Redhad et al. [9] também destacam que as APIs são cruciais para a integração de sistemas, servindo como contratos que definem como os componentes de *software* devem interagir. Além disso, um dos grandes valores das APIs é a capacidade de reutilizar o código, permitindo que os desenvolvedores construam softwares com mais complexidade de maneira mais eficiente ao aproveitar o trabalho já realizado por outros. Esta reutilização não apenas acelera o processo de desenvolvimento, mas também garante que as melhores práticas e soluções testadas sejam incorporadas nos novos projetos [7].

A importância das APIs na arquitetura de software moderna é inegável [1] [10]. Elas desempenham um papel fundamental na construção de aplicações distribuídas, permitindo que diferentes serviços e componentes se comuniquem de forma eficiente e majoritariamente segura [9]. Com a ascensão de arquiteturas baseadas em microsserviços [11] e a proliferação de dispositivos conectados na Internet das Coisas (IoT, do inglês *Internet of Things*), as APIs tornaram-se ainda mais críticas para o funcionamento eficaz dos sistemas [12].

2.2 APIs Web

As APIs Web representam um subconjunto específico de APIs operando predominantemente sobre protocolos web, como HTTP ou HTTPS, projetadas para serem consumidas por aplicações web, dispositivos móveis e outros sistemas. Diferentemente das APIs tradicionais, as APIs Web são acessíveis remotamente e frequentemente utilizam formatos padronizados, como JSON ou XML, para comunicação. Enquanto as APIs RESTful são um exemplo notório de APIs Web, existem outros estilos arquiteturais, como SOAP e GraphQL, demonstrando a versatilidade e a ampla aplicação das APIs no ambiente digital moderno [13]. Neste trabalho, nos concentraremos nas APIs REST, que serão detalhadas

a seguir.

2.2.1 APIs REST

O paradigma REST (do inglês, *REpresentational State Transfer*) estabelece um conjunto de princípios arquiteturais que têm se mostrado fundamentais para o desenvolvimento de sistemas distribuídos na web [1]. As APIs que seguem esses princípios são frequentemente denominadas APIs *RESTful*. Essas interfaces são organizadas em uma coleção de serviços Web *RESTful*, cada um encarregado de gerenciar operações *CRUD* (Criar, Ler, Atualizar, Deletar do inglês, *Create*, *Read*, *Update*, *Delete*) sobre determinados recursos [1].

Neste cenário, um "recurso" é uma entidade que pode ser disponibilizada na web, podendo ser tão variada quanto um vídeo, uma imagem ou um pedido de compra. Estes recursos são comumente acessíveis através de Identificadores de Recursos Uniformes (URIs, do inglês *Uniform Resource Identifier*), o que as torna localizáveis e manipuláveis através de protocolos de aplicação, como o *HTTP* (do inglês, *HyperText Transfer Protocol*). Cada URI que aponta para um ou mais desses recursos é conhecido como um *endpoint* de API [1].

As APIs *RESTful* adotam um conjunto padronizado de diretrizes para a implementação de métodos *HTTP*, conforme descrito a seguir [1]:

- **GET/Ler**: utilizado para acessar e recuperar informações sobre um recurso ou conjunto de recursos;
- **POST/Criar**: empregado para a criação de novos recursos;
- **PUT/Alterar**: serve para modificar um recurso previamente existente;
- **DELETE/Deletar**: aplicado para a remoção de um recurso específico;

Por exemplo, considere uma API REST que gerencia uma biblioteca. Uma possível URI HTTP apontando para o recurso, poderia ser **'`/livros`'**.

Neste caso, a operação *HTTP* **'`GET /livros`'** é usada para recuperar a lista de todos os livros e **'`POST /livros`'** poderia ser utilizado para adicionar um novo livro na biblioteca.

A URI do recurso e os métodos *HTTP* podem aceitar parâmetros de entrada para especificar informações adicionar para a execução das operações da API, como o identificador do livro a ser recuperado (por exemplo, '`/livros/{idLivro}`') ou um objeto estruturado para ser adicionado à coleção usando o método POST.

2.3 Ameaças à Segurança de APIs Web

Com a evolução da arquitetura de software moderna, as APIs Web têm sido incorporadas em diversas soluções tecnológicas. Devido à sua ampla utilização e ao fato de estarem entre os principais vetores de ataque [10], evidencia-se a relevância de se adotar padrões de segurança rigorosos no desenvolvimento e manutenção dessas interfaces [3] [10].

A segurança das APIs Web é comprometida por uma série de vulnerabilidades potenciais, muitas das quais são destacadas no OWASP API Security Top 10 [14]. A OWASP (do inglês, *Open Web Application Security Project*) é uma organização sem fins lucrativos que se dedica a melhorar a segurança dos softwares e, devido à sua abordagem aberta e colaborativa, as informações e ferramentas que ela fornece são amplamente reconhecidas pela comunidade global de segurança da informação. O documento da OWASP descreve as dez principais vulnerabilidades de segurança de APIs. Essas vulnerabilidades abrangem desde falhas em mecanismos de autenticação e autorização até a exposição indevida de dados sensíveis. Tais fragilidades não apenas ameaçam a integridade dos sistemas que dependem desses serviços, mas também podem resultar em violações de dados e impactos negativos nas operações comerciais [3]. A lista a seguir apresenta as dez principais vulnerabilidades de segurança de APIs, conforme definido pela OWASP [14] em 2023.

- **API1:2023 Broken Object Level Authorization**: trata-se de situações em que os *endpoints* de API são suscetíveis à manipulação do ID de um objeto enviado na solicitação, permitindo que atacantes acessem ou modifiquem objetos aos quais não deveriam ter acesso;
- **API2:2023 Broken Authentication**: vulnerabilidades surgem quando os mecanismos de autenticação de uma API são insuficientes ou mal implementados, tornando-os alvos fáceis para ataques;
- **API3:2023 Broken Object Property Level Authorization**: destaca-se pela exposição inadequada de propriedades de um objeto através de *endpoints* de API, particularmente em APIs REST e GraphQL;
- **API4: 2023 Unrestricted Resource Consumption**: surge quando uma API não impõe limites adequados no consumo de recursos, como largura de banda, CPU, memória e armazenamento;
- **API5:2023 Broken Function Level Authorization**: representa situações em que uma API permite que usuários não autorizados acessem funções ou recursos específicos;

- **API6:2023 Unrestricted Access to Sensitive Business Flows**: relaciona-se à exposição de fluxos de negócios críticos sem restrições de acesso adequadas;
- **API7:2023 Server Side Request Forgery**: evidencia-se quando um *endpoint* da API busca um recurso remoto com base em URLs fornecidas pelo usuário, sem validação adequada;
- **API8:2023 Security Misconfiguration**: manifesta-se quando uma API opera com configurações de segurança inadequadas;
- **API9:2023 Improper Inventory Management**: resulta da falta de controle e documentação sobre as versões e *endpoints* de uma API;
- **API10:2023 Unsafe Consumption of APIs**: surge devido à ausência de controle e documentação adequados sobre as versões e *endpoints* de uma API;

Além da lista específica para segurança de APIs, a OWASP também elaborou o renomado "Top 10" que destaca as principais vulnerabilidades de segurança em aplicações web em geral. Embora não seja focada em APIs, esta lista é relevante para o contexto deste trabalho, pois muitas das vulnerabilidades listadas também podem ser exploradas através de APIs. A lista a seguir apresenta as dez principais vulnerabilidades de segurança em aplicações *web*, conforme definido pela *OWASP* [15] em 2021.

- **A1:2021 Broken Access Control**: mecanismos de autenticação de uma API insuficientes ou mal implementados facilitam ataques;
- **A2:2021 Cryptographic Failures**: uso de algoritmos criptográficos fracos ou implementação incorreta de algoritmos robustos;
- **A3:2021 Injection**: possibilidade de injeção de código malicioso, como SQL, *Cross-Site Scripting (XSS)* e injeção de comandos;
- **A4:2021 Insecure Design**: decisões de design que levam à falta de validação de entrada, ausência de mecanismos de autenticação, autorização e uso de componentes vulneráveis;
- **A5:2021 Security Misconfiguration**: configurações inadequadas, como padrões inseguros, armazenamento em nuvem mal configurado e definições incorretas de segurança do servidor web;
- **A6:2021 Vulnerable and Outdated Components**: uso de componentes com vulnerabilidades reconhecidas;

- **A7:2021 Identification and Authentication Failures**: falhas na identificação e autenticação, como senhas fracas, métodos inseguros e falta de proteção contra-ataques de força bruta;
- **A8:2021 Software and Data Integrity Failures**: falhas na integridade de dados e software, como ausência de validações e proteção contra-ataques de repetição;
- **A9:2021 Security Logging and Monitoring Failures**: ausência de mecanismos adequados de monitoramento e *logging*;
- **A10:2021 Server-Side Request Forgery**: *endpoints* da API que buscam recursos remotos baseados em URLs do usuário sem validação apropriada;

2.4 Testes de segurança

Considerando a vasta gama de possíveis vulnerabilidades e os riscos inerentes associados ao desenvolvimento de software, a implementação de testes de segurança emerge como uma prática importante. Essa abordagem proativa é essencial para identificar e mitigar falhas potenciais antes que elas se transformem em problemas reais, garantindo assim a integridade, a disponibilidade e a confidencialidade dos sistemas e dados. Ao antecipar as vulnerabilidades através de testes rigorosos, os desenvolvedores podem prevenir uma série de ameaças e fortalecer as defesas de suas aplicações contra ataques mal-intencionados.

Existe uma variedade de abordagens diferentes para testes de segurança, cada qual com suas vantagens e desvantagens, sendo que nenhuma será perfeita. Em um alto nível, existem três abordagens principais, testes do tipo *white box* (do inglês, caixa branca), *black box* (do inglês, caixa pret) e *gray box* (do inglês, caixa cinza) [16]. A diferença entre eles é definida pelo nível de acesso, do testador, aos recursos relacionados ao software. Em um dos extremos, *white box* requer um acesso completo, ao código fonte, diagramas de design, etc. Em um outro extremo, *black box* implica pouco ou nenhum acesso aos recursos. Enquanto que em testes do tipo *gray box* existe um acesso parcial a recursos como documentação [17][16].

2.4.1 Caixa Branca

Testes que utilizam a abordagem *white box*, também conhecido como *clear-box testing* (do inglês, caixa transparente), teste estrutural ou teste baseado em código, tem como base a análise das estruturas internas de uma aplicação ou sistema [17] [16]. Nesse tipo de teste, o testador tem acesso à estrutura interna, design e implementação do código-fonte, arquitetura e outros detalhes técnicos do aplicativo ou sistema em teste [17] [16].

O teste de caixa branca pode ser realizado em diferentes níveis do ciclo de vida do desenvolvimento de software (SDLC, do ingles *Software Development Life Cycle*),

incluindo testes unitários, testes de integração e testes de sistema [17]. Ele pode ser executado usando diferentes técnicas, como análise estática e análise dinâmica [17].

A análise estática envolve uma revisão do código-fonte ou do código binário de uma aplicação sem executá-la, utilizando ferramentas como revisores de código e verificadores de sintaxe. O objetivo é identificar defeitos e vulnerabilidades antes que a aplicação seja implantada [17].

A análise dinâmica, por outro lado, envolve a análise do comportamento de uma aplicação enquanto está em execução, utilizando ferramentas como depuradores, perfis de desempenho e monitores de desempenho. Tem como objetivo identificar defeitos e vulnerabilidades que podem aparecer apenas durante a execução [17].

2.4.2 Caixa Preta

Testes do tipo *black box*, ou caixa preta, implicam que o testador tem conhecimento apenas do que pode observar externamente no comportamento do sistema ou aplicativo em teste. Este tipo de teste é orientado pelos resultados visíveis das ações do usuário, sem considerar como o software processa as entradas e produz as saídas. A ênfase está em validar a funcionalidade e conformidade do sistema com os requisitos especificados e em identificar quaisquer desvios ou defeitos comportamentais do ponto de vista do usuário final [17] [18].

Dentre as formas de teste do tipo caixa preta, destacam-se [17] [18]:

- Testes de invasão: uma técnica em que o testador age como um atacante e tenta encontrar e explorar vulnerabilidades no sistema ou aplicação sem ter conhecimento prévio do seu funcionamento interno.
- Testes aleatórios, incluindo fuzzing: onde o software é submetido a entradas aleatórias ou inválidas para verificar se ele pode lidar de forma segura e sem falhas.
- Ferramentas de proxy: pode ser usadas para interceptar e modificar solicitações e respostas HTTP, permitindo que o testador examine o tráfego entre o navegador e o servidor e identifique vulnerabilidades.
- Ferramentas de varredura automática: podem ser usadas para identificar vulnerabilidades em aplicativos web sem a necessidade de interação manual do testador.

2.4.3 Caixa Cinza

No meio do caminho entre os testes de caixa branca e caixa preta, estão os testes *gray box*, ou caixa cinza. Nesse contexto o testador tem conhecimento parcial do aplicativo. Nesse caso, informações sobre entrada do usuário, controles de validação de entrada e

armazenamento de dados podem ser conhecidas pelo testador. O objetivo é verificar como a entrada do usuário é processada pelo aplicativo e armazenada no sistema de back-end [17].

2.5 Fuzzing

Fuzzing, abreviação de *fuzz testing* é uma técnica automatizada de teste de software que emprega uma grande diversidade de entradas, tanto normais como anormais, para a aplicação alvo [19][20]. Introduzido inicialmente por Miller et al. em 1988 [21], o processo envolve não apenas o fornecimento dessas entradas à aplicação, mas também o monitoramento contínuo dos estados de execução do software para identificar comportamentos inesperados, falhas ou travamentos. Esse monitoramento permite que os desenvolvedores e testadores detectem e corrijam vulnerabilidades que só se manifestam quando o software é submetido a condições extremas ou atípicas [20]. Em contraste com outras técnicas, o *fuzzing* destaca-se pela facilidade de implantação, extensibilidade e aplicabilidade, podendo ser executado com ou sem acesso ao código-fonte [19]. Adicionalmente, devido à sua natureza baseada na execução real do *software*, o *fuzzing* apresenta alta precisão [19].

A ferramenta que viabiliza essa técnica é denominada *fuzzer*. Atuando como um gerador de entradas, ele cria os dados de teste e injeta no programa alvo. Existem diferentes abordagens para a geração de testes, como os métodos baseados em mutação e gramática [6] — métodos de mutação modificam entradas de teste já existentes, enquanto técnicas baseadas em gramática geram novas entradas a partir de uma descrição estrutural da entrada. Além disso, os *fuzzers* podem operar em abordagens que vão desde a caixa-preta [5], sem conhecimento do código, até a caixa-branca, com total acesso ao código-fonte [20].

O panorama atual do desenvolvimento de *fuzzing* reflete um crescente interesse na abordagem de segurança de software. Nos últimos anos, a adoção de técnicas de *fuzzing* mostra um aumento constante [20] [19] [22] e tem sido reconhecida por sua eficácia para descobrir tanto falhas de implementação, quanto vulnerabilidades de segurança. Além disso, o *fuzzing* tem se integrado com outras técnicas avançadas, como algoritmo genético, análise de contaminação e execução simbólica [20]. A comunidade acadêmica demonstra um interesse crescente no assunto, como evidenciado pelo aumento nas publicações sobre *fuzzing* [3]. Este foco contínuo sugere que o *fuzzing* continuará a ser uma técnica valiosa no campo da segurança de software nos próximos anos [22].

2.5.1 Técnicas de Fuzzing

Existem diversas técnicas que foram introduzidas ao contexto de *fuzzing* para melhorar a geração de entradas aleatórias, no trabalho de Chen *et al.* [23] é feito um

levantamento das principais técnicas:

1. Técnicas de Geração de Amostras:
   São empregadas para escolher e alterar sementes [1], além de limitar e criar novas amostras. Podem ser categorizadas em três grupos principais:

   - *Random mutation* (mutação randômica): a ideia principal é utilizar mutação de algum campo para gerar uma semente que será utilizada para gerar novas entradas. Sistemas de *fuzzing* que tem como base essa técnica são rapidamente implementadas e possuem alta escalabilidade, já que não computacionalmente exigentes. Porém, essa técnica é ineficiente para encontrar problemas complexos, já que não tem visão do estado de execução da aplicação e não possui algum tipo de parâmetro eficiente para as mutações.
   - *Grammar representation* (representação gramatical): em contraste com a mutação randômica, essa técnica utiliza uma gramática – que pode ser feita automaticamente com técnicas como análise dinâmica ou manualmente – para gerar entradas mais direcionadas. É eficiente para aplicações com formatos de entrada mais complexos, mas necessita de um conhecimento do sistema para a gramática.
   - *Scheduling algorithms* (algoritmos de agendamento): são métodos que tem como objetivo maximizar a saída do processo de *fuzzing*, com otimizações de escolha de estratégias para sementes e estratégia de mutação.

2. Técnicas de análise dinâmica:
   Tem como base a extração de informações dinâmicas a partir de um software em execução, podendo ser divididas em:

   - *Dynamic symbolic execution* (execução dinâmica simbólica): consiste em é uma determinar possíveis entradas de um programa usando valores simbólicos como entradas e gerando novos caminhos com base em restrições simbólicas coletadas.
   - *Coverage feedback* (feedback de cobertura): utiliza informações sobre quais partes do código de um programa foram executadas durante os testes para orientar a geração de casos de teste na próxima iteração. Isso resulta em uma cobertura de caminho aprimorada e aumenta a probabilidade de encontrar vulnerabilidades. É mais eficiente do que o *fuzzing* aleatório devido ao seu mecanismo de feedback.
   - *Dynamic taint analysis*: observa e identifica a propagação explícita e o uso inadequado de dados influenciados ou alterados por um invasor ou código ma-

[1] Refere-se a uma entrada inicial ou caso de teste que é usado como base para gerar novas entradas

licioso na memória de um programa. Essa técnica utiliza dados de entrada com "rótulos"para especificar como o programa manipula esses dados de entrada e quais partes do programa foram afetadas pelos dados contaminados. Essa técnica pode ser combinada com outras estratégias para aprimorar a precisão do *fuzzing*.

3. Outros
   Uma outra técnica é o uso de aprendizado de máquina em técnicas de *fuzzing*. Aprendizado de máquina pode ser usado para melhorar a eficiência e eficácia do processo de *fuzzing*, gerando melhores casos de teste e melhorando a cobertura de código. Também tem utilidade para identificar caminhos e seções potencialmente perigosos no programa, aprendendo com caminhos e seções em muitos programas que escondem vulnerabilidades. Outra maneira de usar o aprendizado de máquina em técnicas de *fuzzing* é ajudar a escolher uma estratégia de mutação após aprender os efeitos do uso de diferentes estratégias de mutação.

2.5.2 Vantagens e Desvantagens

Sendo uma ferramenta, das diversas que devem ser utilizadas para uma análise de segurança robusta, *fuzzing* tem vantagens e desvantagens, dentre as quais podemos citar:

Vantagens:

- Pode ser relativamente simples e fácil de implementar [23];
- Tem aplicação em uma ampla variedades de programas e sistemas [23][4][6];
- Pode ser automatizado para executar testes repetitivos e de longa duração [23];
- Tem a capacidade de detectar vulnerabilidades e *bugs* que não são facilmente identificáveis por outras técnicas de segurança [23];
- *Fuzzing* pode ser utilizado para testar sistemas de software em tempo real, permitindo que problemas sejam corrigidos rapidamente [24];

Desvantagens

- Pode ser difícil gerar casos de teste que explorem todas as possíveis vulnerabilidades [23][25]
- Pode produzir muitos resultados de teste falsos positivos ou falsos negativos [23].
- Pode ser difícil determinar se um resultado de teste é uma vulnerabilidade conhecida ou uma vulnerabilidade desconhecida [23].

- Pode ser difícil determinar a causa raiz de uma vulnerabilidade identificada pelo *fuzzing* [23].

2.6 Fuzzing de APIs REST

Para realizar o *fuzzing* de API REST, os testadores podem utilizar ferramentas especializadas que capturam e reproduzem o tráfego HTTP, analisam solicitações e respostas HTTP e realizam o processo de *fuzzing* com heurísticas predefinidas ou regras definidas pelo usuário [4][6]. Algumas ferramentas também podem aproveitar especificações Swagger[2] para orientar o *fuzzing* e gerar casos de teste mais inteligentes.

Considerando que APIs REST possuem um formato de entrada que pode ser variado, *fuzzing* baseado em gramática se torna uma boa alternativa em comparação com entradas puramente aleatórias [6]. Não só isso, mas também requer a seleção cuidadosa de casos de teste e monitoramento do feedback dinâmico do serviço [4]. Ao combinar essas técnicas, os testadores podem alcançar uma cobertura de código elevada e detectar uma ampla gama de bugs e vulnerabilidades de segurança [4].

2.6.1 Ferramentas de fuzzing de APIs REST

Nos últimos anos, diversos avanços foram alcançados no campo do *fuzzing* de APIs REST, revelando ferramentas inovadoras e estratégias eficazes para a identificação de vulnerabilidades. Dentre as ferramentas disponíveis, destacam-se:

- **bBOXRT** É uma abordagem e ferramenta que tem como objetivo avaliar a robustez de serviços REST. A ferramenta usa um documento de descrição de serviço como entrada para gerar um conjunto de entradas inválidas que são enviadas ao serviço em combinação com parâmetros válidos. Além disso, a ferramenta pode operar como um proxy de injeção de falhas entre o cliente e o servidor. As respostas do serviço são analisadas preliminarmente em busca de casos suspeitos de falha e armazenadas para análise detalhada posterior pelo usuário da ferramenta [27].

- **EVOMASTER** Ferramenta de código aberto que usa técnicas evolutivas para gerar casos de teste de nível de sistema para aplicativos empresariais e web. Ele é projetado para ser uma ferramenta de caixa branca (mas também tem uma funcionalidade de caixa preta), o que significa que leva em consideração os detalhes internos do código do lado do servidor. Atualmente, ele se concentra em APIs RESTful em execução em JVMs, mas pode ser estendido para outras linguagens e contextos de

[2] Swagger permite a descrição da estrutura de APIs em um formato legível por máquinas, facilitando a geração automática de documentação interativa e bibliotecas de cliente em várias linguagens. Isso é alcançado ao fazer com que a API retorne um arquivo YAML ou JSON que adere à Especificação OpenAPI, detalhando operações, parâmetros, autorizações e informações adicionais sobre a API [26].

teste de sistema. É composto por dois componentes principais: um processo principal responsável pelas funcionalidades principais e um driver de processo que inicia, para e reinicia o sistema em teste e instrumenta seu código-fonte. A ferramenta é capaz de encontrar falhas em projetos de código aberto e é uma alternativa para testes manuais e outras ferramentas de geração de testes [28].

- **ResTest** RESTest é uma ferramenta de teste de caixa-preta de código aberto para APIs RESTful que suporta várias técnicas, incluindo fuzzing. Ela recebe como entrada a especificação da API em formato OAS e gera, e opcionalmente executa, casos de teste usando técnicas como *fuzzing*, teste aleatório adaptativo e baseado em restrições. RESTest depende de geradores de dados de teste personalizados para gerar automaticamente dados realistas, como endereços de e-mail e códigos de idioma. O framework pode ser facilmente estendido com novos geradores, técnicas de geração e escritores de teste. RESTest já se mostrou útil na detecção automatizada de bugs do mundo real em APIs comerciais usadas por milhões de usuários em todo o mundo [29].

- **RestCT** Utiliza testes combinatórios para gerar casos de teste automaticamente. A ferramenta é capaz de testar as interações de um número específico de operações, bem como as interações de parâmetros de entrada específicos em cada operação. A ferramenta é totalmente automática e pode ser usada com qualquer especificação Swagger [30].

- **Restler** É a primeira ferramenta de fuzzing automático e inteligente para APIs REST. Ele analisa uma especificação Swagger, infere dependências entre tipos de solicitação e gera testes definidos como sequências de solicitações que satisfazem essas dependências. A ferramenta aprende dinamicamente quais sequências de solicitações são válidas ou inválidas, analisando as respostas do serviço a esses testes. REST-ler é capaz de encontrar novos bugs em serviços REST, incluindo vulnerabilidades de segurança, e é uma ferramenta promissora para testar a segurança e confiabilidade de serviços em nuvem [4].

- **RestTestGen** Utiliza uma abordagem de caixa-preta automatizada para APIs RESTful. Ele gera casos de teste para cenários de execução nominal e de erro, com o objetivo de detectar defeitos de implementação usando dois oráculos distintos. A ferramenta usa o grafo de dependência de operações para modelar explicitamente as dependências de dados entre as operações, e aplica operadores de mutação para gerar valores de entrada adicionais para cada operação. A validação empírica da ferramenta em 87 APIs RESTful reais revelou um grande número de defeitos de implementação [31].

- **Schemathesis** Possui testes baseados em propriedades para encontrar erros semânticos e falhas em APIs da web OpenAPI ou GraphQL. A ferramenta é capaz de fornecer cobertura de código por meio de instrumentação no lado do servidor ou suporte para testes em processo. Em uma avaliação comparativa com outras oito ferramentas, Schemathesis superou todas as outras em encontrar defeitos e lidar com serviços-alvo sem erros internos fatais. A ferramenta é mantida como um projeto de código aberto pelos desenvolvedores e pode ser usada em vários casos de uso no desenvolvimento de software [32].

- **Pythia** Combina *fuzzing* baseado em gramática com feedback guiado por cobertura e mutações baseadas em aprendizado para fuzzing de API REST com estado. Ele gera casos de teste gramaticalmente válidos e prioriza os casos de teste que são mais propensos a encontrar bugs. A ferramenta foi avaliada experimentalmente em três serviços em nuvem de código aberto em escala de produção, mostrando que supera abordagens anteriores em cobertura de código e encontrou 29 novos bugs, que estão sendo relatados aos proprietários dos serviços [6].

- **foREST** Usa uma abordagem baseada em árvore para gerar casos de teste e detectar bugs e vulnerabilidades. A ferramenta modela as dependências de APIs com uma estrutura de árvore, o que reduz a complexidade das dependências e captura a prioridade das dependências de recursos. A ferramenta foi avaliada em experimentos com serviços REST do mundo real e superou outras ferramentas de *fuzzing* de API RESTful em termos de cobertura de código e desempenho. A ferramenta está disponível como código aberto no GitHub [33].

- **Miner** Tem um sistema híbrida de dados para melhorar o desempenho na descoberta de erros profundos e bugs de segurança. A ferramenta implementa três novos designs que trabalham juntos para abordar as limitações dos fuzzers de API REST existentes. MINER utiliza um modelo de rede neural para melhorar a qualidade da geração de solicitações em um modelo de sequência e um verificador de regras de segurança baseado em dados para capturar erros causados por parâmetros indefinidos. A abordagem é genérica e pode ser aplicada a maioria dos fuzzers de API REST [34].

REFERÊNCIAS

[1] SEGURA, S. et al. Metamorphic Testing of RESTful Web APIs. *IEEE Transactions on Software Engineering*, v. 44, n. 11, p. 1083–1099, nov. 2018. ISSN 1939-3520. Conference Name: IEEE Transactions on Software Engineering.

[2] IDRIS, M.; SYARIF, I.; WINARNO, I. Development of Vulnerable Web Application Based on OWASP API Security Risks. In: *2021 International Electronics Symposium (IES)*. [S.l.: s.n.], 2021. p. 190–194.

[3] DíAZ-ROJAS, J. A. et al. Web API Security Vulnerabilities and Mitigation Mechanisms: A Systematic Mapping Study. In: *2021 9th International Conference in Software Engineering Research and Innovation (CONISOFT)*. [S.l.: s.n.], 2021. p. 207–218.

[4] ATLIDAKIS, V.; GODEFROID, P.; POLISHCHUK, M. *REST-ler: Automatic Intelligent REST API Fuzzing*. arXiv, 2018. ArXiv:1806.09739 [cs]. Disponível em: <http://arxiv.org/abs/1806.09739>.

[5] TSAI, C.-H.; TSAI, S.-C.; HUANG, S.-K. *REST API Fuzzing by Coverage Level Guided Blackbox Testing*. arXiv, 2021. ArXiv:2112.15485 [cs]. Disponível em: <http://arxiv.org/abs/2112.15485>.

[6] ATLIDAKIS, V. et al. *Pythia: Grammar-Based Fuzzing of REST APIs with Coverage-guided Feedback and Learning-based Mutations*. arXiv, 2020. ArXiv:2005.11498 [cs]. Disponível em: <http://arxiv.org/abs/2005.11498>.

[7] THAYER, K.; CHASINS, S. E.; KO, A. J. A Theory of Robust API Knowledge. *ACM Transactions on Computing Education*, v. 21, n. 1, p. 8:1–8:32, jan. 2021. Disponível em: <https://dl.acm.org/doi/10.1145/3444945>.

[8] SERVICES, I. A. W. *What is an API? - Application Programming Interfaces Explained*. 2023. Acessado em: 2023-08-30. Disponível em: <https://aws.amazon.com/what-is/api/>.

[9] HAT, I. R. *What is an API?* 2023. <https://www.redhat.com/en/topics/api/what-are-application-programming-interfaces>. Acessado em: 2023-08-30.

[10] IDRIS, M.; SYARIF, I.; WINARNO, I. Web Application Security Education Platform Based on OWASP API Security Project. *EMITTER International Journal of Engineering Technology*, p. 246–261, dez. 2022. ISSN 2443-1168. Disponível em: <https://emitter.pens.ac.id/index.php/emitter/article/view/705>.

[11] WANG, X. et al. Exploring Efficient Microservice Level Parallelism. In: *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. [S.l.: s.n.], 2022. p. 223–233. ISSN: 1530-2075.

[12] AL-MASRI, E. Enhancing the Microservices Architecture for the Internet of Things. In: *2018 IEEE International Conference on Big Data (Big Data)*. [S.l.: s.n.], 2018. p. 5119–5125.

[13] LAURET, A. *The Design of Web APIs*. [S.l.]: Manning, 2019. ISBN 978-1-61729-510-2.

[14] FOUNDATION, O. *API Security Project*. 2023. <https://owasp.org/www-project-api-security/>. Acessado em: 2023-09-02. Disponível em: <https://owasp.org/www-project-api-security/>.

[15] FOUNDATION, O. *OWASP Top Ten*. 2023. <https://owasp.org/Top10/>. Acessado em: 2023-09-02. Disponível em: <https://owasp.org/Top10/>.

[16] SUTTON, M.; GREENE, A.; AMINI, P. *Fuzzing: Brute Force Vulnerability Discovery*. [S.l.]: Addison-Wesley Professional, 2007.

[17] WSTG - v4.2 | OWASP Foundation. Disponível em: <https://owasp.org/www-project-web-security-testing-guide/v42/>.

[18] SHAHBAZI, A.; MILLER, J. Black-Box String Test Case Generation through a Multi-Objective Optimization. *IEEE Transactions on Software Engineering*, v. 42, n. 4, p. 361–378, abr. 2016. ISSN 1939-3520. Conference Name: IEEE Transactions on Software Engineering. Disponível em: <https://ieeexplore.ieee.org/document/7293669>.

[19] LI, J.; ZHAO, B.; ZHANG, C. Fuzzing: a survey. *Cybersecurity*, v. 1, n. 1, p. 6, jun. 2018. ISSN 2523-3246. Disponível em: <https://doi.org/10.1186/s42400-018-0002-y>.

[20] LIANG, H. et al. Fuzzing: State of the Art. *IEEE Transactions on Reliability*, v. 67, n. 3, p. 1199–1218, set. 2018. ISSN 1558-1721. Conference Name: IEEE Transactions on Reliability.

[21] MILLER, B. P.; FREDRIKSEN, L.; SO, B. An empirical study of the reliability of UNIX utilities. *Communications of the ACM*, v. 33, n. 12, p. 32–44, dez. 1990. ISSN 0001-0782. Disponível em: <https://dl.acm.org/doi/10.1145/96267.96279>.

[22] BOEHME, M.; CADAR, C.; ROYCHOUDHURY, A. Fuzzing: Challenges and Reflections. *IEEE Software*, v. 38, n. 3, p. 79–86, maio 2021. ISSN 1937-4194. Conference Name: IEEE Software.

[23] CHEN, C. et al. A systematic review of fuzzing techniques. *Computers & Security*, v. 75, p. 118–137, 2018. ISSN 0167-4048. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0167404818300658>.

[24] GODEFROID, P.; HUANG, B.-Y.; POLISHCHUK, M. Intelligent REST API data fuzzing. In: *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2020. (ESEC/FSE 2020), p. 725–736. ISBN 978-1-4503-7043-1. Disponível em: <https://dl.acm.org/doi/10.1145/3368089.3409719>.

[25] ZHANG, M.; ARCURI, A. Open Problems in Fuzzing RESTful APIs: A Comparison of Tools. *ACM Transactions on Software Engineering and Methodology*, v. 32, n. 6, p. 144:1–144:45, set. 2023. ISSN 1049-331X. Disponível em: <https://dl.acm.org/doi/10.1145/3597205>.

[26] SWAGGER. *What Is Swagger*. 2023. <https://swagger.io/docs/specification/2-0/what-is-swagger/>. Acessado em: 2023-11-20. Disponível em: <https://swagger.io/docs/specification/2-0/what-is-swagger/>.

[27] LARANJEIRO, N.; AGNELO, J.; BERNARDINO, J. A Black Box Tool for Robustness Testing of REST Services. *IEEE Access*, v. 9, p. 24738–24754, 2021. ISSN 2169-3536. Conference Name: IEEE Access. Disponível em: <https://ieeexplore.ieee.org/document/9344640>.

[28] ARCURI, A. EvoMaster: Evolutionary Multi-context Automated System Test Generation. In: *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. [s.n.], 2018. p. 394–397. ArXiv:1901.04472 [cs]. Disponível em: <http://arxiv.org/abs/1901.04472>.

[29] MARTIN-LOPEZ, A.; SEGURA, S.; RUIZ-CORTéS, A. RESTest: automated black-box testing of RESTful web APIs. In: *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. New York, NY, USA: Association for Computing Machinery, 2021. (ISSTA 2021), p. 682–685. ISBN 978-1-4503-8459-9. Disponível em: <https://dl.acm.org/doi/10.1145/3460319.3469082>.

[30] WU, H. et al. Combinatorial testing of RESTful APIs. In: *Proceedings of the 44th International Conference on Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2022. (ICSE '22), p. 426–437. ISBN 978-1-4503-9221-1. Disponível em: <https://dl.acm.org/doi/10.1145/3510003.3510151>.

[31] VIGLIANISI, E.; DALLAGO, M.; CECCATO, M. RESTTESTGEN: Automated Black-Box Testing of RESTful APIs. In: *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. [s.n.], 2020. p. 142–152. ISSN: 2159-4848. Disponível em: <https://ieeexplore.ieee.org/document/9159077>.

[32] HATFIELD-DODDS, Z.; DYGALO, D. Deriving Semantics-Aware Fuzzers from Web API Schemas. In: *2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. [s.n.], 2022. p. 345–346. ISSN: 2574-1926. Disponível em: <https://ieeexplore.ieee.org/document/9793781>.

[33] LIN, J. et al. *foREST: A Tree-based Approach for Fuzzing RESTful APIs*. arXiv, 2022. ArXiv:2203.02906 [cs]. Disponível em: <http://arxiv.org/abs/2203.02906>.

[34] LYU, C. et al. *MINER: A Hybrid Data-Driven Approach for REST API Fuzzing*. arXiv, 2023. ArXiv:2303.02545 [cs]. Disponível em: <http://arxiv.org/abs/2303.02545>.